Trace-Driven Performance Evaluation of Distributed File Systems under Enterprise Application Workload

Roman Talyansky
SAP AG, SAP Research
Ra'anana, Israel
roman.talyansky@sap.com

Bernd Scheuermann
SAP AG, SAP Research
Karlsruhe, Germany
bernd.scheuermann@sap.com

Björn Kolbeck
Zuse Institute Berlin
Berlin, Germany
kolbeck@zib.de

Jan Stender
Zuse Institute Berlin
Berlin, Germany
stender@zib.de

***Abstract*—We evaluate the performance of distributed file systems (DFS) under the IO load produced by enterprise applications. The evaluation is carried out using a trace-based approach that allows for recording file system accesses of an application and re-playing the trace on an arbitrary file systems under test. This method avoids the overhead that is associated with installing and maintaining complex multi-tier enterprise applications. Our trace-based assessment suggests that DFS, and specifically XtreemFS have a good potential to support transactional IO load in distributed environments: they demonstrate good performance of read operations and scalability in general, while at the same time opening the way for significant TCO reduction. We found, however, that more work should be done to improve the performance of write operations, which is crucial in transactional enterprise applications.**

I. Introduction

Evaluating the performance impact of a Distributed File System (DFS) in enterprise application scenarios can be a complicated and costly exercise. Typically, enterprise applications are transactional, database-centric applications that require a complex software stack. One has to invest significant efforts in installation, configuration and tuning of the application prior to the performance evaluation. Performing such an evaluation on multiple candidate DFSs clearly multiplies the effort. It may be the case that the evaluation turns out to be negative after a lot of effort has been already invested.

In this paper, we describe a trace-based methodology for performing such an evaluation without having to actually run the application. The methodology is based on collecting traces of IO operations that the application generates while running over a file system. The trace-based methodology allows us to evaluate performance of multiple candidate DFSs under a workload generated by enterprise applications while minimizing the large overhead associated with installing and maintaining the complex software stacks that are typical of these applications. Using traces we can evaluate the performance of the application in various flexible conditions, such as varying user activity, without having to install the application.

In recent years, various advanced distributed file systems were presented which claim to offer high scalability along with the chance of reducing Total Cost of Ownership (TCO) (see e.g. [1], [2], [3], [4]). In our search for candidate file systems, we decided to focus on such object-based distributed file systems which are commonly based on the following principles:

1) Multiple data servers, often named Object Storage Devices (OSDs): A set of storage nodes stores data from the clients. Multiple data servers may allow for parallel access and redundant data storage. The overall bandwidth and operations can be aggregated and is less limited by some dedicated storage controller throughput rather by aggregative network bandwidth. Furthermore, the sum over all the caches of individual OSDs provides a distributed cache of the distributed file system.
2) Object-based file systems may be built from plenty cheap commodity storage devices. This can offer a noticeable economical advantage over commonly used centralized filer technologies consisting of comparatively expensive hardware components.
3) Most of currently available commercial storage systems enable *scale up*, where to support applications with growing storage demand, the customer needs to acquire more powerful but very expensive storage hardware (possibly replacing the existing solution). In contrast to such filer systems, object-oriented distributed file systems allow for taking advantage of *scale out* effects, where to enhance the existing file system, more commodity storage hardware is added, and the file system seamlessly and transparently incorporates it according to its load balancing scheme.
4) Software based fault tolerance: When a system is composed of a large number of cheap nodes, node failures become more likely. Therefore, the DFS is commonly required to handle such failures on a software basis. Note that fault tolerance is a desired feature, but it is not evaluated withing this work.
5) Parallel access: Good persistent layer designs involve a large number of parallel data servers to handle the workload. The performance of file IO can benefit from striping of data objects and replica management. Replica also contribute to higher availability and fault resilience.

The goal of this work is two-fold. First, we will introduce a practical trace-based methodology for facilitating the procedure of file system evaluations. Secondly, we aim to evaluate

the performance of object-oriented distributed file systems serving the workload produced by transactional enterprise applications from SAP.

The remainder of this paper is structured as follows. In Section II, we provide the description of the file systems that we chose for our study. Then Section III describes the enterprise applications that were used as the application workload. Section IV introduces the trace-based methodology that was used in our study. In Section V, we describe our simulations of the selected file system's distribution functions. Section VI describes the experiments that we performed and presents the experimental results. We summarize our work and give an outline on future work in Section VII.

II. Related Work

Distributed and parallel file systems have undergone a development of several decades. The first milestone were file systems that rely on a centralized server. One of the oldest and most prominent such file systems is NFS [5], a single-server file system that allows users to mount and access different volumes. For large amounts of data and users, however, the single-server setup soon emerged as a performance bottleneck. To overcome the limitations of NFS with large-scale installations, multi-server file systems like AFS [6], [7] and later Coda [7] were introduced that partition data among many file servers. NFS as well as AFS/Coda store file contents together with file metadata. However, AFS differs from NFS in its handling of file changes. AFS offers a session semantics. During a session, a client accesses a local copy of a file. A server-side callback mechanism notifies clients of concurrent changes by other clients. The session ends when the file is closed, which causes changes to be transmitted back to the server. Coda enhances AFS with a transactional session model, which allows clients to continue even when disconnected from the server. Changes are eventually reconciled when the server becomes available again.

With the advent of storage area networks (SANs), parallel block-based file systems like GFS [8], OCFS2 [9] and GPFS [10] were developed. Such file systems offer clients direct access to block devices in a SAN via Fibre Channel or iSCSI and coordinate concurrent parallel access to these devices. Typically, clients are deeply involved in the management of blocks as well as metadata on the disk storage devices. GFS stripes files across multiple disks that are aggregated through a distributed logical volume manager. This guarantees scalability in terms of storage capacity and access performance, as multiple disk devices can be accessed in parallel through multiple paths. To further increase scalability, GPFS partially offloads the block management from the clients to network storage device servers.

In recent years, the principle of object-based storage [11], [12] has replaced the traditional concept of block-based storage as the prevalent design pattern for distributed and parallel file systems. Object-based file systems split file data into file metadata (directory tree, file name, size) and file content. The metadata is stored on one or more metadata servers and the file content is stored on one or more storage servers. This design allows file IO operations to be executed in parallel on several storage servers. Often, the storage servers are off-the-shelve servers rather than SAN-based storage which can reduce storage cost.

Lustre [4] and Panasas ActiveScale [3] are two popular commercial object-based file systems. Both, Lustre and Panasas, use regular off-the-shelve servers for metadata and storage servers. The target for both file systems in high performance computing and both support special networking hardware such as Infiniband. In contrast to Lustre, Panasas ActiveScale does support RAID across multiple storage servers for fault-tolerance.

CEPH [2] and XtreemFS [1] are two object-based file systems developed within research projects; both are open source projects. CEPH is a traditional parallel file system for cluster environments. XtreemFS is a WAN-distributed file system for grid and cloud infrastructures but also supports parallel I/O. We will use these two file systems for our experiments.

CEPH uses a cluster of metadata servers (MDS) and object storage devices (OSD). A directory tree can be partitioned and stored on several MDS servers, thereby load balancing [13] and avoiding a single metadata server bottleneck. The file content is distributed among all OSDs in the cluster and each file can be striped across one or more OSDs for parallel IO. In contrast to other distributed file systems CEPH does not use a list of OSDs per file to locate file content. Instead, it uses a mechanism similar to consistent hashing so that each client can locate file content by using a cluster map and a unique file identifier [14]. The cluster map contains all OSDs in the system and needs to be consistent on all client nodes. While this mechanism reduces load on the metadata server, it requires data to be moved across OSDs when the system is scaled, i.e. OSDs are added. CEPH has a POSIX compliant semantics. For a single client access, IO operations are buffered and cached. However for multi-client access, all IO operations are synchronous to the OSDs. The replication in CEPH employs a master/slave scheme with a master OSD which disseminates the updates to the slave OSDs. The master is defined by the hash function.

The focus of XtreemFS is to support operations over wide-area networks and to cope with limited bandwidth, message loss and delay as well as host crashes and datacenter failures. As an object-based file system, XtreemFS stores the directory tree on the Metadata and Replica Catalog (MRC) and file content on OSDs. The MRC uses an LSM-tree based database which can handle volumes that are larger than the main memory [15]. OSDs can be added to the system as needed without any data re-balancing; empty OSDs are automatically used for newly created files and replicas. XtreemFS has a POSIX compliant semantics, also for striped replicas [16]. File replication is supported in two modes, read-only and read-write. Read-only replication handles the distribution of file content for immutable files. This mode avoids the overhead for coordinating replica consistency and supports a large number

of replicas and also partial replicas. In contrast, the read-write replication allows files to be modified and is fully POSIX compatible. XtreemFS uses a master/slave replication with decentralized lease coordination for failover[17], [18]. The decentralized lease coordination allows XtreemFS to operate without a centralized lock service which would severely limit scalability[19].

Similar to the object-based file systems, Google's GFS [20], HDFS [21] and PVFS/PVFS2 [22] separate metadata from file content. In contrast to the object-based design, these systems keep a block index per file on the metadata server. This allows these file systems to be more flexible where to place file data. However, this comes at the price that clients need to contact the metadata server on each new block. Google's GFS alleviates this problem by using a very large block size of 64MB by default.

File system traces have been used since the 1980s to analyze and simulate file system behavior [23], [24]. TraceFS [25] is a low-overhead stackable file system that intercepts calls at the Linux VFS layer. TraceFS may incur elapsed time overhead above 12% [25], which is intrusive. For this reason in our study we use SystemTap [26] to intercept IO accesses, as we explain later.

TABLE I
IO CHARACTERIZATION OF THE MAXDB WORKLOAD

	distr	number	avg size	overall size
read	2.3%	14389	6884	99M
write	48%	302201	7995	2.416G
lseek	49.7%	312752		
metadata ops	0.004%	2627		

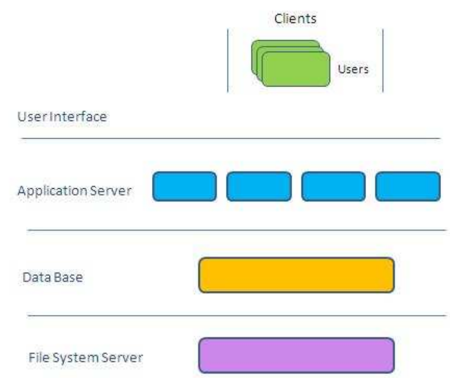

Fig. 1. Typical multi-tier system environment of a transactional enterprise application.

III. Enterprise Applications

This section describes applications that we used to characterize IO load of enterprise applications. We also point out potential benefits that an object-based distributed file systems (DFS) can provide for the applications.

A. Transactional Application

Typically, the design of enterprise solutions is based on a multi-tier architecture as visualized in Figure 1. Such an architecture is comprised of multiple clients communicating with the business logic executed on a range of application servers. When the application server needs to perform a database transaction on behalf of the application, it uses the database as persistent storage. The database ensures transactional consistency of the data stored in the underlying file system. For the large majority of file operations, the enterprise application accesses the file system through the centralized database. Therefore we chose to represent the transactional application by the IO load that the database generated while serving the transactional enterprise application. In our experiments, we used the relational MaxDB database from SAP [27]. Load on the database is produced by a Sales benchmark executing on MaxDB. Table I provides a characterization of the workload MaxDB generates in terms of IO operations during the initialization phase of the application where file operations are largely write-oriented.

We acknowledge the fact that a transactional load is typically read-oriented. However, for the purpose of demonstrating the advantages our trace-based approach, the choice of the specific trace can be arbitrary. Nevertheless, further work is required in order to better quantify the performance impact of DFS under transactional load. Note that since the application opens all its files at the beginning and closes at the end, the number of the metadata operations is close to zero.

B. Enterprise Search Application

The enterprise search engine TREX [28] is used as the second enterprise application for the experiments where the engine is executed in indexing and query mode. In the indexing mode, TREX processes a commonly large collection of documents and constructs the search index. TREX reads documents in parallel, generating parallel read load on the underlying file system. Then TREX calculates the search index and writes it back to the file system in parallel, resulting in a collection of index files that are written onto the file system imposing parallel write load. Thus the index creation mode of TREX generates IO load that is characterized as parallel massive read and write loads. Running in query mode, TREX uses the created index to search for documents accordingly to the submitted search request.

IV. Trace-based Modeling of File System Workload

Typical enterprise application environments are very complex systems, which may be hard to install, to maintain and to use directly in performance testing. So it is desirable to develop an approach to capture the application behavior as IO model which can be replayed as performance benchmark without actually executing the application. The IO model can also be used to analyze the IO access patterns of the application. Knowing the application access patterns can help choosing the most appropriate storage system and tuning it to the application needs. The trace-based methodology consists

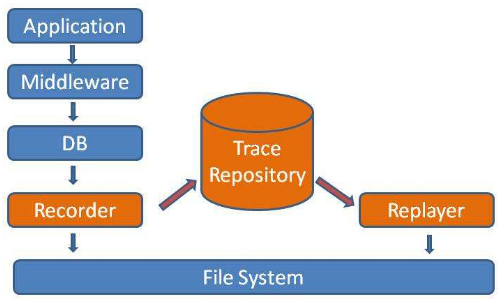

Fig. 2. Trace-based methodology.

TABLE II
DETAILS OF THE RECORDED IO OPERATIONS

IO Op	File Desc	Offset	Number Bytes	Resp Time
open	✔			✔
close	✔			✔
read	✔		✔	✔
write	✔		✔	✔
lseek	✔	✔		✔

of two steps: trace recording and trace replay as shown in Figure 2.

A. Trace Recording

To create the IO model, we suggest running the real enterprise application over a reference file system thereby intercepting and recording the trace of IO accesses generated by the application. We refer to the captured application trace as *IO stream*. To intercept IO accesses, we used SystemTap [26]. SystemTap allows the users to incorporate probes into a stand-alone kernel module that is loaded into the kernel. When loaded, the system can trigger user's probe handlers in order to collect and pass data.

Table II summarizes the recorded information for each IO operation. Each IO access in the trace is associated with the identifiers of the issuing thread and process. The start and end time of each operation is recorded allowing to calculate the response time of the operation. Time intervals between any pair of consecutive IO operations along the same thread are also calculated. We assume that those time intervals characterize the application's 'think time' which the application spends for CPU-bound calculations, synchronization events, networking, etc.

B. Trace Replay

The trace can be replayed over a certain file system under test or over a simulation model of a filesystem. The trace replay step is structured as follows.

First the original directory tree is created which contains the files accessed during replay. Since the contents of the original data streams of IO operations and the original database files need not be preserved, dummy random data is written to dummy files which contain random contents.

Subsequently the replay procedure re-creates threads and processes which in turn re-produce the file IO operations previously recorded. For this, the replayer exploits the thread and process identifications recorded in the trace.

The recorded 'think time' is used to reproduce the application behavior during the replay step in simulations and file system evaluations. Note that the trace interception mechanism is not aware of the application internal inter-dependencies, i.e. whether a particular read operation appears only after a particular write operation. However, the IO replayer always executes IO operations in their original order per thread.

Since trace interception mechanism is not aware of the the internal application inter-dependencies, the replayer does not keep any synchronization between IO operations across different threads. Also we are not able to take into account the effects that running the application over different environments could have changed the behavior of the application itself. For example different response times of the file system could cause different order or timing of the IO accesses produced by the application.

Replaying application IO traces eliminates the need to use the real application for running the experiments. A further benefit of the proposed trace-based methodology is the reproducibility of the IO load – we are able to reproduce the same IO load across our experiments with different file systems.

V. SIMULATION

The purpose of simulation is to further simplify performance assessment of (enterprise) applications running over DFS. The simplification is twofold, as we use the trace instead of the real application and furthermore, the file system is simulated. Since DFS is a complex system we decided to simulate only one its inherent component - the *distribution function* of the DFS - the component that defines the distribution of the application IO accesses to OSDs and in turn affects the IO performance of the file system. When an application accesses a file at a certain offset, the distribution function of the underlying DFS maps this access to a set of OSDs that store the stripes and replicas of the accessed portion of the file.

To make the quality of the distribution function measurable we suggest to calculate the standard deviation of the number of application accesses to OSDs for each second of the replayed trace. Specifically, for each second and for each IO operation type, i.e. *read*, *write*, we count the number of times each OSD is accessed under the application IO load. As the result we get for each IO operation type and each second in the application execution trace an *OSD pattern* - an array of counts, where each entry in the array corresponds to one OSD. At this stage we calculate the standard deviation of each array as the quality measure of the distribution function. A low standard deviation signifies a well-balanced usage of the OSDs. Finally

we calculate the average over the values of standard deviation for the entire trace. Thus at the end of this process we end up with two numbers - the average standard deviation for *read* and *write* operations.

Above we described the process of simulation-based performance assessment for a single IO stream. As described earlier, we synthesize a workload of multiple IO streams by setting the start time of each IO stream randomly. The start time of each IO stream was randomly chosen from the interval $[0, 300sec]$ according to the uniform distribution. The choice of the start times was done only once. All the simulations and experiments used that choice. Since the total run time of each experiment and for each file system was in the interval $[1.5, 3]$ hours, the parallel execution of IO streams took the majority of the time span in each experiment.

Figures 3 and 4 show the simulation results as a function of the number of IO streams, when we used the traces of MaxDB application. In the figures and in the discussion below, SW denotes stripe width of the file system volume and in our experiments SW is equal the number of OSDs available for the file system.

To interpret the simulation results we provide the characterization of the application OSD patterns (single IO trace). An entry in an OSD pattern is *active* if its value is large - above some threshold. All the other entries are *inactive*. Table III shows the distribution of OSD patterns for a single IO stream for 0, 1 and 2 active entries and the threshold value of 30, which was chosen empirically. It turns out that the distribution of OSD patterns is not very sensitive to the threshold value. The purpose of Table III is to show that in a single IO stream the majority of OSD patterns are imbalanced and have up to two active entries. Consequently as more IO streams are superimposed, the number of active entries in OSD patterns grows. Directly from the definition of standard deviation it follows that standard deviation of an OSD pattern gets its maximum when the pattern is *half full*, namely when the number of active entries is around the half of SW. This result can be seen from analyzing the following statement for standard deviation of an OSD pattern with k active entries:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{k}(m-\mu)^2 + \sum_{i=k+1}^{N}(0-\mu)^2}{N}},$$

where N is the number of OSDs in the OSD pattern, m is the value of an active entry and μ is the mean over the values of the entries in the OSD pattern, i.e. $\mu = \frac{km}{N}$. The value of m is assumed to be constant. Analytical study of the above expression shows that σ achieves the maximum at $k = 0.5N$

This fact combined with the evidence from Table III explains the raise of the standard deviation for write operations with the increase in the number of IO streams until the number of half full OSD patterns is maximized. After this point, when the number of IO streams continues to grow, the number of half full OSD patterns drops and the average standard deviation for the write operation also drops and starts to approach gradually 0.

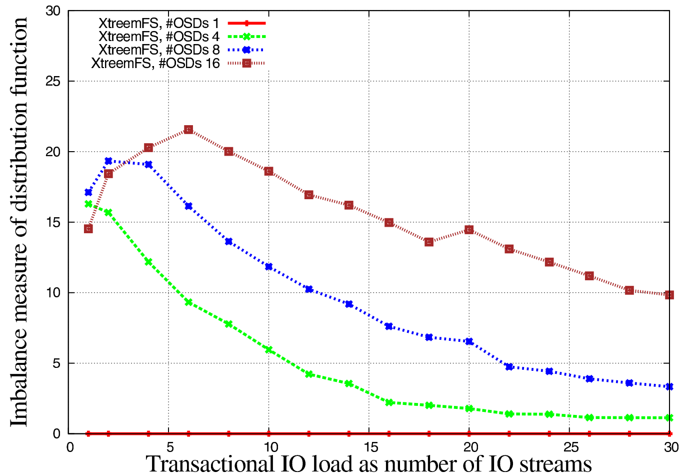

Fig. 3. Distribution function evaluation - write operations.

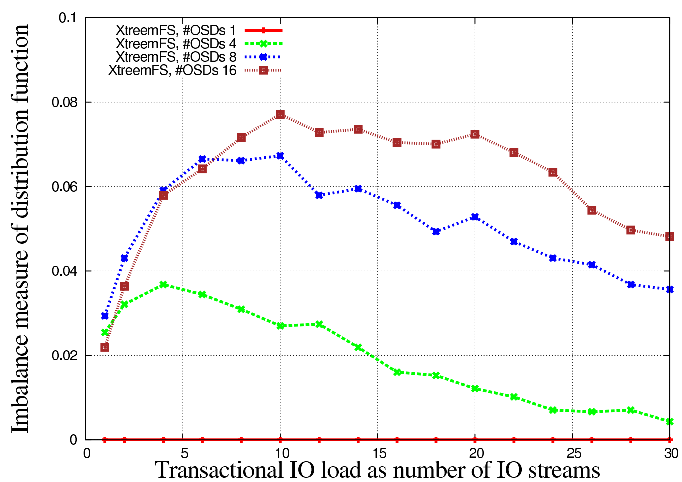

Fig. 4. Distribution function evaluation - read operations.

The analysis of Figure 3 reveals that as the number of IO streams grows, smaller SW of the underlying file system volume enables to distribute application IO accesses better than larger SW. However, for file systems volumes with smaller SW, the load on each OSD may also grow faster, which can subsequently affect the performance of the file system. We conclude, therefore that the choice of SW should be set as a compromise of two requirements: the requirement to avoid resource under-utilization due to the imbalanced OSD usage on the one hand and the performance requirements of the resulting file system on the other hand.

Figure 4 shows the simulation results for read operations in MaxDB application. As with write operations, the average standard deviation for read operations initially increases until the number of half full patterns is maximized and then approaches 0 as the number of IO streams continues to grow.

VI. EXPERIMENTS

In this section, we describe the experiments that were performed on the candidate file systems. As we stated earlier we are interested in evaluating distributed file systems which

TABLE III
CHARACTERIZATION OF OSD PATTERNS FOR WRITE OP

# active entries	SW=4	SW=8	SW=16
0	16.75%	16.75%	16.75%
1	55.7%	55.65%	55.65%
2	12.64%	12.56%	12.52%

follow the object-based file system design, allow high availability, performance and scaling out on a commodity hardware. Second, since we seek ways to lower TCO, we further narrow our attention to open source candidates. Thus for our analysis we selected two distributed file systems: XtreemFS version 1.0 and CEPH version 19.1. XtreemFS uses only fuse client. In our experiments CEPH used the kernel client. The replication level in the both systems was 1. XtreemFS also attracts our attention because it supports operations over wide area networks. This feature may be useful for example in a federated enterprise search use case, where to shorten the latency of a document load operation, the collection of documents may be replicated close to the searching clients. In addition, we included an NFS server and an NAS device in our experiments for comparison. NAS devices are currently the de-facto standard in industry as enterprise storage backends. Therefore, we included an NAS device from a leading storage vendor in our experiments as a baseline. Similarly to NFS servers, these NAS devices can be scaled-up (up-to a certain limit) by using more expensive devices with larger main-memory caches and larger storage capacity. In our experiments, the NAS device of a size about 15 TB was connected to the clients via Ethernet using an NFS protocol.

All the experiments used the MaxDB application to generate the IO load. Each experiment was repeated 3 times and the average over results is reported in this paper. To provide the insight on the stability of the results we calculated the standard deviation over the three repetitions of each experiment and normalized it by the mean latency over three experiments. Then for each number of IO streams we calculated the average over standard deviations over SW 1, 4 and 8 and summarized the results in Table IV. The data in the table shows that the standard deviation is high for a small number of IO streams and it decreases to acceptable values as the number of IO streams increases. To shed light on this standard deviation behavior Table V shows read latency for 3 experiments along IO streams. One can see that for small number of IO streams the first experiment has a significantly higher latency, giving rise to the high average latency for the small number of IO streams reported in Table IV. This phenomenon may be explained by either space allocation on disk by the underlying file system at the first experiment, which may add significant overhead as compared to the second and third experiment or an artifact of the java JIT, which gradually compiles XtreemFS code during the first experiment.

For the MaxDB application, we used latency and aggregative throughput as performance measures. The average latency was measured over all accesses in one experiment. Aggregative

TABLE IV
AVERAGE STANDARD DEVIATION FOR READ AND WRITE OPERATIONS ALONG THE NUMBER OF IO STREAMS.

	Number of IO streams						
op	**1**	**2**	**4**	**6**	**8**	**10**	**12**
read	1.091	1.089	0.164	0.263	0.066	0.105	0.183
write	0.862	0.880	0.042	0.057	0.104	0.072	0.053

TABLE V
WRITE LATENCY IN SECONDS FOR DIFFERENT EXPERIMENTS AND IO STREAMS.

	Number of IO streams						
Exp	**1**	**2**	**4**	**6**	**8**	**10**	**12**
1	0.063	0.071	0.045	0.073	0.099	0.127	0.178
2	0.011	0.011	0.054	0.071	0.060	0.174	0.183
3	0.020	0.011	0.048	0.055	0.052	0.127	0.183

throughput was measured over all the concurrent IO traces in each individual experiment. For the TREX application, we choose to use application-oriented performance measure - the overall time to construct the index.

At the recording stage for the MaxDB application, we used a state of the art filer as the file system. In contrast to MaxDB application, whose IO traces we recorded and replayed, we ran TREX application directly over XtreemFS and NFS file systems.

A. Experimental Environment

As the testbed we used a 15 nodes cluster with SuSE Linux Enterprise 11 installed at each node. Each node has 2GB of RAM, two cores with frequency 2.4GHz and cache size 4Mb. The nodes are connected via a 1Gb/s Ethernet network. The throughput performance of the disks in the cluster nodes is around 125MB/s. Performing the tests at the same dedicated cluster environment without using virtualization layers ensures accuracy, reproducibility and fairness of the comparison. In our experiments we used release 1.1 of the XtreemFS from 2009-09-18.

For our XtreemFS experiments, we created a number of volumes with 1, 4 and 8 OSDs. For each volume, the stripe width (SW) equals the number of OSDs. For MaxDB application, to emulate concurrent transactional load, we ran the trace replay simultaneously on several nodes, one trace per node. For each SW (#OSDs), we ran the experiments with the following number of concurrent IO streams: 1, 2, 4, 6, 8, 10 and 12. Figures 5, 6, 7 and 8 present the experimental results, which we discuss in the next sub-section.

B. Results

Figures 5, 6, 7 and 8 present the experimental results for the baseline filer technology, NFS, XtreemFS volumes with #OSDs 1, 4 and 8 and preliminary results for CEPH with #OSDs 4, when we used our MaxDB application to generate the IO load. Figures 5 and 6 present the read latency and throughput results. To interpret them, Table VI provides the details on the latency results for three experiments and one

TABLE VI
READ LATENCY IN MICROSECONDS FOR NFS AND XTREEMFS (XFS)

#exp	NFS	XFS, #OSDs=1	XFS, #OSDs=4	XFS, #OSDs=8
1	47926	240690	85619	37074
2	15.1	26572	4471	3976
3	15.2	16286	5827	5198

IO stream. Examining the table one can see that at the first experiment NFS reads the data from the disk, while at the two subsequent experiments it serves the read requests from the cache. This explains the excellent read throughput of NFS for one IO stream. Similar phenomenon takes place for XtreemFS. Pay attention, however, to two observations: XtreemFS cannot serve all read requests from the cache and at the first experiments, when the data is read from the disk, the read latency of XtreemFS improves with the rise in #OSDs, since XtreemFS distributes disk access operations among several OSDs. When the number of IO stream grows, the read latency of NFS and XtreemFS with #OSDs 1 starts to rise, while this rise is less prominent or absent for XtreemFS with #OSDs 4 and 8. This phenomenon is explained by the network connectivity that becomes the bottleneck - in the case of NFS and XtreemFS with #OSDs 1, all read operations from all the nodes access the same node. In contrast, the clients of XtreemFS with #OSDs 4 and 8 access the serving OSDs directly so that the network burden is distributed among several OSD nodes, reducing the resulting latency.

Note also that for #OSDs 4 and 8, when the number of IO streams grows from 1 to 2 and 4, the read latency of XtreemFS even improves. We explain this phenomenon by utilization of OSD-based distributed cache of XtreemFS on the one hand and overlapping working sets of separate simultaneous IO streams on the other hand. When an IO stream first accesses a stripe of the overlapping part, it reads the stripe from the disk. When the subsequent IO streams access the same stripe, they find it already in the cache, eliminating the need to access the disk, which shortens the average access latency across all the concurrent IO streams. More simultaneous IO streams result in shorter average latency. This explains the latency improvement in case of 4 and 8 OSDs when the number of OSDs grows from 1 to 2 and 4. For higher number of IO streams, the network connectivity becomes the bottleneck as we explained earlier.

Since we strove to collocate IO streams with OSD nodes, as long as the number of IO streams is at most as #OSDs, each IO stream is collocated with an OSD node. In this case due to the random striping policy of XtreemFS, the portion 1/#OSDs of all IO accesses of each IO stream are handled locally by each node. However, each additional IO stream beyond #OSDs is not collocated with any OSD node and thus is more expensive in terms of latency, giving the rise of the latency. For this reason the average latency of read and write operations is increased when the number of IO streams is beyond #OSDs.

Analyzing the preliminary results for CEPH, we note that CEPH achieves the best read latency among all our file system candidates for 2 IO streams. We also note the latency improvement as the number of IO streams increases from 1 to 2. We attribute the both phenomena to caching the reads on the one hand and low probability to access the same file for small number of IO streams.

Figure 7 shows the write latency of the baseline filer technology, NFS volume, XtreemFS volumes with #OSDs 1, 4 and 8 and the preliminary latency results under CEPH with #OSDs 4. All files in those experiments were open with O_SYNC flag, blocking the calling process until the data has been transferred to to the hard disk by Linux. We set this flag since in a typical transactional application when a transaction is committed, the commit operation is written to the disk or non-volatile memory to insure its persistence. Since the filer used non-volatile memory for the write operations, its latency is far below the latencies of the other file systems. As with the latency of the read operations, the latency of write operations of NFS shows bad scalability with growing number of IO streams, while the latency of write operations of XtreemFS volumes is more resilient for the growing number of IO streams. Furthermore, it may be observed that a higher #OSDs provides with a better resilience to the growing number of IO streams. Our interpretation of this observation is that with growing #OSDs (and respectively the number of OSDs) XtreemFS distributes the synchronous write operations among several OSDs and thus effectively balances the write load and reduces the collisions at each OSD when several write operations are executed in a narrow time window.

The preliminary results of the experiments with CEPH show that while its write latency is low for 1 IO stream, the growth rate of the write latency is high as compared to the other file system candidates.

Figure 8 shows write throughput in our experiments. As in the case of read operations, the graphs of throughput of write operations provide with the mirror picture of the write latency. It is interesting to observe that for each #OSDs 1, 4, and 8, at XtreenFS the saturation of the rise in the throughput occurs at progressively higher levels. Actually for #OSDs 8, the saturation is not approached even for 12 IO streams.

Note, also, that with #OSDs 4, XtreemFS approaches the throughput of the baseline filer and with #OSDs 8 exceeds it in spite of the fact that the filer used non-volatile memory to shorten latency. This clearly shows the advantage of the distributed XtreemFS file system to scale out with the commodity hardware, as opposed to the scale-up capability of a typical filer technology.

Next we provide the experimental results of constructing TREX index over NFS and XtreemFS file systems. The results are shown in Figure 9. In the experiments with TREX we chose an application-oriented measure - the total time required to construct the index. Note that #OSDs in the figure is shown for XtreemFS only. Examining Figure 9 one can see that NFS provides better results than XtreemFS with #OSDs 1. However when #OSDs of XtreemFS grows, XtreemFS distributes the application IO load among several OSDs, and consequently

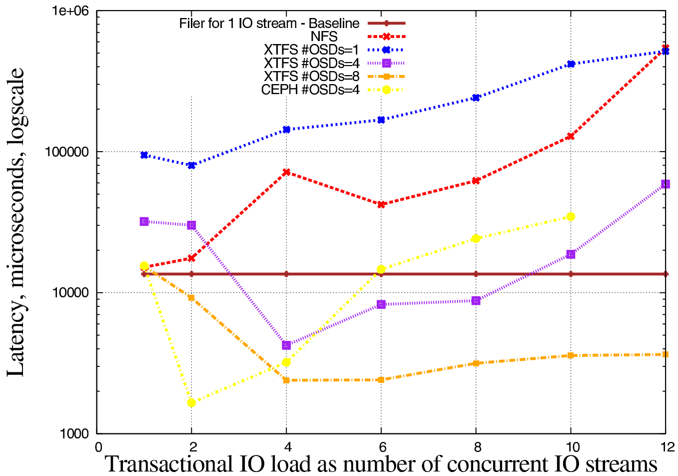

Fig. 5. Read latency of transactional application under several file systems for varying IO load.

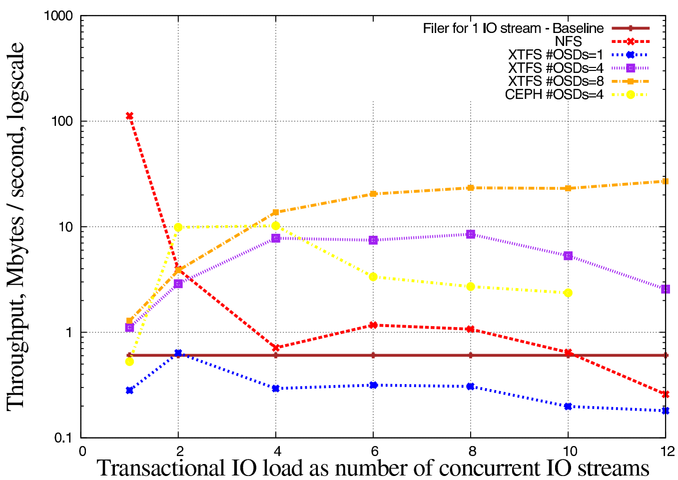

Fig. 6. Read throughput of transactional application under several file systems for varying IO load.

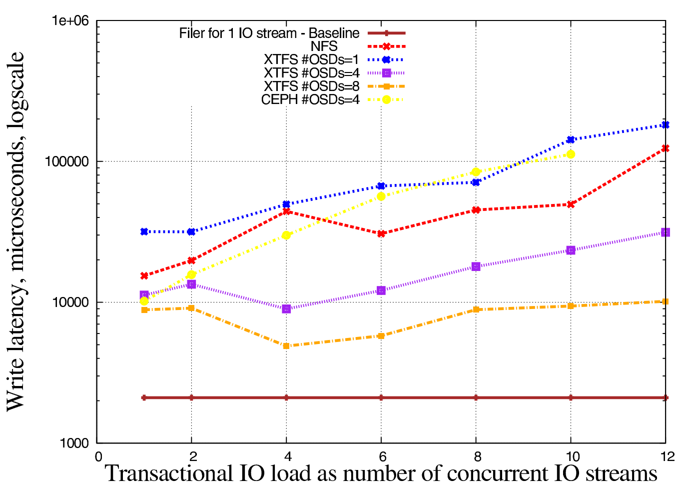

Fig. 7. Write latency of transactional application under several file systems for varying IO load.

achieves better performance results.

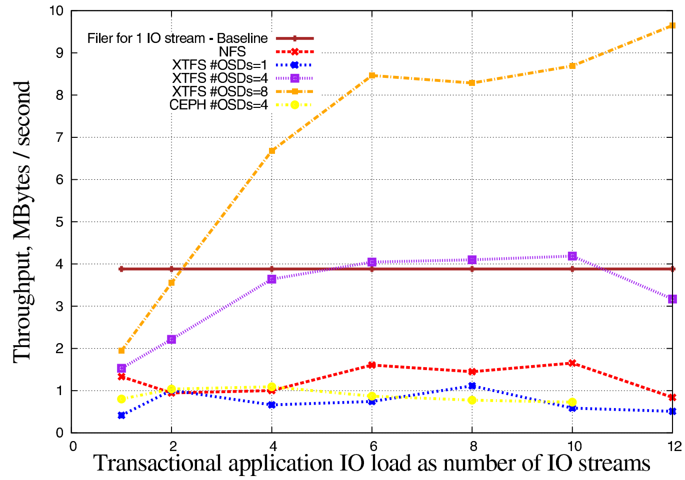

Fig. 8. Write throughput of transactional application under several file systems for varying IO load.

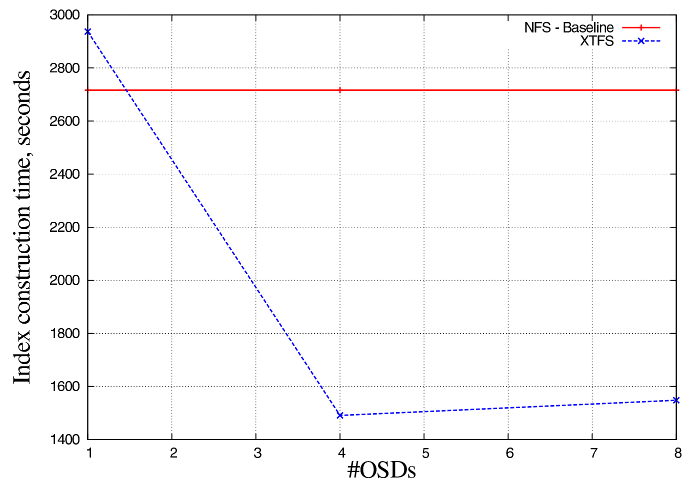

Fig. 9. Enterprise search application: index construction time.

VII. Conclusions and Future Research

We performed the performance analysis measurements which show promising results that support the following conclusions:

- XtreemFS provides good latency and throughput of read operations under transactional application workload. The latency of write operations is, however, far above the required level. This indicates that more work should be done to lower the latency of synchronous write operations, i.e. by means of attaching battery backup to conventional RAM devices or using solid state drives.
- XtreemFS enables several clients to access OSD nodes directly, eliminating the communication bottleneck of accessing the single file server. This architectural change helps in keeping latencies low and getting high throughput of IO operations. Thus XtreemFS enables scaling of the number of concurrently executed application instances (i.e., IO streams) without sacrificing performance.
- When an application uses synchronous write operations,

XtreemFS effectively distributes those write operations over several OSDs keeping low latencies and getting high throughput. It shows the scalability of XtreemFS when facing the growing rate of synchronous write operations.

- While the performance of CEPH is excellent for small number of IO streams, the growth rate of its latency (and decrease rate of throughput) is higher than for the other file system candidates.
- XtreemFS and CEPH may help to reduce TCO of running business applications: they scales out over commodity hardware and their performance scales almost linearly with stripe width.
- DFS parameters such as stripe width and stripe size should be tuned for the expected enterprise application load to achieve the required performance on the one hand and to avoid under-utilization of the storage resources on the other hand.

Our experiments show the potential of XtreemFS to support transactional load effectively in terms of low TCO, good performance of read operations and supporting scalability of the transactional load. However more work should be done to reduce the latency of synchronous write load, which is crucial for serving transactional enterprise applications.

During the experiments it turned out that the proposed trace-based methodology is a useful means for performance evaluation of DFS under transactional load that eliminates the need to install, tune and maintain the real application. It enables to generate reproducible IO load and to apply it to several distributed file systems. Simulating the distribution function of DFSs we were able to easily assess the load balancing among different OSDs.

In the future we plan to collect traces from additional application phases and use them to evaluate the DFS candidates. We will study where the lazy I/O data integrity extention of POSIX semantics is applicable at our business applications, apply it at the appropriate locations and measure the performance gain of this optimization for file systems that support this extension. Furthermore, we plan to complement our simulation of the file system's distribution function. We also plan to broaden the simulation to validate more aspects of DFS which may affect the performance of applications running over the file system. An additional direction of our further research is to study the conditions and limitations of using a superposition of several instances of a single-tenant IO stream to emulate multi-tenancy schemes. We will identify multi-tenancy schemes for which this approach is suitable along with the required adjustments to the trace recorder and replayer. Further research will also examine the benefits of replication capabilities with respect to high availability and performance of transactional enterprise applications.

VIII. Acknowledgments

We would like to thank the following people for their contributions to the paper: Adolf Hohl for sharing with us his insightful thoughts; Eliezer Levy for providing valuable comments and reviewing the paper; Sage Weil and Yehuda Sadeh Weinraub for their support in using CEPH file system; Roman Magarshak and Ruediger Winter for their dedicated support in maintaining the underlying cluster.

References

[1] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "XtreemFS: a case for object-based storage in Grid data management," in *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB 2007*, 2007.

[2] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *OSDI*, 2006, pp. 307–320.

[3] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu, "The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage," in *SC*, 2004, p. 53.

[4] Lustre, Inc., "Lustre architecture whitepaper."

[5] B. Nowicki, "NFS: Network file system protocol specification," in *RFC 1094*, 1989.

[6] J. H. Howard, M. L. Kazar, S. G. Menees, A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 51–81, 1988.

[7] M. Satyanarayanan, "Scalable, secure, and highly available distributed file access," *Computer*, vol. 23, no. 5, pp. 9–18, 20–21, 1990.

[8] S. R. Soltis, T. M. Ruwart, M. T. O'Keefe, and M. T. Okeefe, "The global file system," in *In Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, 1996, pp. 319–342.

[9] M. Fasheh, "OCFS2: The oracle clustered file system, version 2," in *Proceedings of the 2006 Linux Symposium*, 2006, pp. 289–302.

[10] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, 2002, pp. 231–244.

[11] M. Mesnier, G. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 8, pp. 84–90, 2003. [Online]. Available: citeseer.ist.psu.edu/mesnier03objectbased.html

[12] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," in *2nd International IEEE Symposium on Mass Storage Systems and Technologies*, 2005.

[13] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004.

[14] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Grid resource management—crush: controlled, scalable, decentralized placement of replicated data," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 122.

[15] J. Stender, K. Björn, M. Högqvist, and F. Hupfeld, "BabuDB: Fast and efficient file system metadata storage," in *6th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 2010)*, 2010.

[16] J. Stender, B. Kolbeck, F. Hupfeld, E. Cesario, E. Focht, M. Hess, J. Malo, and J. Martí, "Striping without sacrifices: maintaining posix semantics in a parallel file system," in *LASCO'08: First USENIX Workshop on Large-Scale Computing*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–8.

[17] B. Kolbeck., "A protocol for efficient and fault-tolerant file replication over wide-area networks," in *TCPP PhD Forum at IPDPS*, 2009.

[18] F. Hupfeld, B. Kolbeck, J. Stender, M. Högqvist, T. Cortes, J. Marti, and J. Malo, "Fatlease: scalable fault-tolerant lease negotiation with paxos," in *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2008, pp. 1–10.

[19] M. K. McKusick and S. Quinlan, "Gfs: Evolution on fast-forward," *Queue*, vol. 7, no. 7, pp. 10–20, 2009.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[21] H. project, "The hadoop file system (hdfs)." [Online]. Available: http://hadoop.apache.org

[22] P. H. Carns, W. B. Lingon III, R. B. Ross, and R. Thakur, "PVFS: a parallel file system for linux clusters," in *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*. Berkeley, CA, USA: USENIX Association, 2000, p. 28.

[23] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A trace-driven analysis of the UNIX 4.2BSD file system," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-85-230, Apr 1985. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1985/5199.html

[24] D. R. Jacob, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *In Proceedings of the 2000 USENIX Annual Technical Conference*, 2000, pp. 41–54.

[25] A. Aranya, C. P. Wright, and E. Zadok, "Tracefs: A file system to trace them all," in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2004, pp. 129–145.

[26] http://sourceware.org/systemtap/.

[27] SAP AG, "SAP MaxDB homepage," URL: http://maxdb.sap.com, 2010.

[28] ——, "Search and classification TREX," https://cw.sdn.sap.com/cw/docs/DOC-22234, 2008.